ECE 4750 Lab 4: Multicore Processor

Basant Amro Khalil (bak94)

December 8, 2021

Section 1. Introduction

The purpose of this lab was to gain some knowledge about topics discussed in class, such as software/hardware co-design, programming single- and multithreaded C programs, and computer architecture evaluation methodologies based on architecture level implementations by designing single and multicore processors. The lab implements significant themes in computer architecture, such as memory networks and incremental design. Some of the components of this lab include an emphasis on structural composition to incrementally create a complex system based on thoroughly unit-tested subsystems, software-hardware co-design so that students should understand how to implement software application, hardware/software interface, and hardware microarchitecture. In the baseline design, we were given a single core processor with its instruction and data cache. In the alternative design, we designed a multicore processor with private instruction caches and a shared, banked data cache.

In addition, we wrote both a single-threaded and multithreaded sorting microbenchmark in C, explored the compiled binary, and ran both programs on both of the designs. The evaluation displays that the alternative design has a lower CPI than the baseline designs for multithreaded programs. For instance, the multithreaded design has a CPI of 1.85 for the multithreaded vvadd test compared to a CPI of 5.45 for the single core design on the same test. For the bsearch test, the alternative design has a CPI of 1.76 compared to 5.05 for the baseline design. In fact, each of the evaluation tests portrays the alternative design with a better CPI than that of the baseline design. These evaluation results emphasize the ability of the alternative design to exploit thread-level parallelism to increase throughput and lower CPI. The alternative design is more compelling than the baseline since it can handle multiple threads simultaneously. Multiple cores enable each core to handle separate streams of data, resulting in better performance for a system that runs multiple applications simultaneously. Multicore processors exploit the thread-level parallelism by allocating different threads within a single program to the different cores on the chip. Ideally, this feature should parallelize an application on p processor resulting in about p* speedup. However, it is not possible to fully parallelize most applications as serial parts of the code can lead to higher execution time. Also, implementing multiple threads can cost significant hardware, area, and energy costs. Despite that, the performance improvement that multicore processors provide, as shown in Table 1, outweigh the costs. Regarding energy use, even though the multicore processor requires more energy due to running multiple threads at one time and supporting some extra hardware, we could claim that the multicore processor saves energy per thread as it allocates less energy for one thread in the end than a single core processor allocates for that thread. Increasing the number of cores will significantly improve performance for systems running multiple concurrent threads. However, not all applications are able to be run in parallel, and the extra overhead in assembly code to deal with the unused cores will slightly slow down the execution.

Section 2. Baseline Design

The baseline design in figure 1 is a single core, pipelined processor with full bypassing composed of an instruction cache and unbanked data cache. The baseline design does not use networks as there are less modules to connect and the memory data will flow through a single processor. In addition, it was easy to manage the val/rdy interfaces between the instruction cache and processor and data cache and processor. The baseline design also uses the alternative design created in lab 2 to reduce stalling through implementing bypassing. Bypassing is critical as it significantly improves the throughput of a processor with few extra hardware, area, and power costs. In addition, it uses the alternative design from lab 3 for the instruction and data caches that can also exploit temporal locality as it is two-way associative. The number of banks is also set to 0 as there is only one core in the baseline. The baseline design is a good design since all instructions executing on one port makes it easier to compare the performance improvements of parallelization after addition of the cores in the alternative design. For software, we also implemented a scalar quicksort algorithm that sorts an array of integers. Quicksort first chooses an element from the array as a pivot and then swaps elements on the right and left sides of the pivot such that elements on the right side are greater than the pivot and elements on the left side are smaller than the pivot. It recursively calls a quicksort algorithm on sides of the pivot to sort the sides individually until the array is sorted. Even though any element of the array could be the pivot, we chose the median value as the pivot for the function. This ensures a good performance of quicksort function with an expected 0(nlog n) performance. Quicksort is a good baseline because it has a good spatial locality in comparison to other sorting algorithms, such as merge sort that divides the array to sort it. We used the quicksort algorithm in the merge sort algorithm in the alternative design to sort the array portions so we can then merge them. As a result, quicksort is a good baseline design since it is used in the merge sort in the alternative design so that we can easier compare the advantages and disadvantages that result from dividing the array among multiple cores. This design, in addition to the baseline design, displays memory hierarchy from the caches, main memory and other levels. Hierarchy implies that there are several different levels in memory that going up levels means better speed and bandwidth and going down the levels means increased capacity. For instance, the cache is at a higher level than the main memory so that it takes less time to access data compared to memory while having lower capacity than the memory levels below it. The main memory is at a lower level than the caches and it has a higher ability to hold more data than the cache but it takes significantly more time to access that data than the levels above it, such as the cache level. The design also implements modularity since the parts of the single core processor were put together using the different processor and cache modules. Modularity refers to a system made of different components that can be connected, which is how this design is designed. Since we didn't have to consider the implementations of the processor and the caches, the design models encapsulation since we can swap our implementation for a different implementation if needed as only the input and output connections should remain the same.

Section 3. Alternative Design

The alternative design is a quad-core processor with four instruction caches and a banked datacache system, shown in figure 7 and in figure 2 with a high level. Networks manage the interface between caches and main memory. The multicore datacache contains four data cache banks, a cachenet, and a memnet by which the cachenet connects the caches and manages the banking of the caches and the memnet connects the banked caches to memory. We connected the multicore datacache, the instruction cache, the memnets which connect to the instruction cache and the four processors. For this design, we used our alternative design for the processor in lab two and the alternative cache design from lab three for the instruction caches. We chose that since the baseline design for the processor in lab two implemented stalling while the alternative design implemented bypassing and stalling and hence using the alternative design might lead to better performance to handle RAW data hazards with less stalling. We also used the alternative design for the instruction caches from lab three instead of the baseline as the alternative design is a two-way set associative cache whereas the baseline design is a direct mapped cache. A two-way set associative cache compared to the direct mapped cache can lead to higher hit rate since an address can have two ways to index to in one set since for any index, there are two ways to store the tag. A larger hit rate means memory is accessed less, which reduces latency. The cachenet, in figure 3, manages the interface between the four processors and the four data cache banks. This network has an upstream (the processor) and the downstream (main memory) message adapter that converts processor and main memory messages to network messages. The upstream adapter extracts the bank bits in Figure 4 from the processor and includes them in the destination header. The Memnet in Figure 5 works similarly as the cachenet, but the upstream adapter inserts a zero always in the destination field. The memnet refills the instruction caches and data cache banks. Finally, the MCoreDataCache module in figure 6 connects the cachenet, the MemNet, and the four cache banks to create the complete four bank data cache system. We instantiate the MCoreDataCache along with four processors, four instruction caches, and MemNet for the instruction caches in the multicore system to be more similar to a realistic processor and cache design system. The incremental approach we followed models modularity and hierarchy since the components are grouped into modules that connect to each other to form the final MultiCore system on top of the hierarchy levels. We implemented a mergesort algorithm to sort an array of integers. The merge sort algorithm combines two sorted arrays into one sorted larger array. The algorithm takes in an argument vector pointer that we use to determine the destination and source array pointers and the first and last index of the array to be sorted. We could determine the size of the array by subtracting the first indeed from the last index. The merge sort algorithm then divides the array into two pieces: one from the first index to the middle index, and one starting from the middle index and ending at the last index. The middle index is calculated by adding half the array size to the first index. We then call the quicksort algorithm on both pieces. The merge sort then uses a for loop to loop over the destination array filling each element as appropriate. The algorithm chooses the value that gets filled into each element of the destination array by comparing the two pieces of the array. Each iteration, the algorithm checks which piece has the smallest value at the "leading" index and that value is then placed at the next index of the destination array. The "leading" index for each subarray starts at the first element and is incremented each time an element of that specific subarray is placed into the destination array. Mergesort is a good implementation as its worst case time complexity performance is O(n logn). It works perfectly to combine two sorted arrays for which we use quicksort to sort. Generally, a multicore processor (four core implemented in the alternative design) is a good design as it can run multiple threads concurrently, but there are some cons to adding more cores. Thus, we can't just add 200 cores, for instance. The reason is listed in the evaluation section below, but it is similar in line to not choosing a 200 pipelined processor in lab two even though the pipelined processor showed better performance than the single cycle. The high level reasoning is that dividing work to be done in parallel then becomes harder with higher overhead. In the first place, we don't need so many cores as not so many tasks need to be executed concurrently in programs usually. Even though the multithreaded merge sort algorithm could be more complex than the single threaded quicksort algorithm implemented in the baseline design, the thread-level parallelism exploited by the multicore processor results in improved performance compared to that of the single core processor. To reiterate, a multicore processor implemented in the alternative design can result in better performance than the single core in the baseline implementation as it provides better bandwidth and higher exploitation of thread-level parallelism.

Section 4. Testing Strategy

Test scripts were provided for this lab, which is intended to test both the hardware and software side of things. The single core system, which has only one processor along with Instruction and data caches is tested with some of the assembly tests that were created as part of lab2. 1-2 instructions are picked up from each category, such as add and mul from rr, addi from rimm, bne from branch, jal from jump, lw and sw from memory, csrr and csrw from csr type of instructions. For the multicore design, the individual design blocks such as CacheNet, MemNet,McoreDataCache are also tested. The CacheNet is tested by connecting test sources, test sinks to the upstream Network and 4-port test memory to the downstream network as shown in Figure 3, similarly the MemNet is tested by connecting test sources, test sinks to the upstream Network and single-port test memory to the downstream network as shown in Figure 5. The McoreDataCache which is a combination of CacheNet, MemNet, four cache banks is also unit tested by connecting test sources, test sinks to the CacheNet and single-port test memory to the MemNet as shown in Figure 6. The test sets used for CacheNet and McoreDataCache are reused from testset in lab3. To accommodate four sources and four sinks, we used TestNetCacheSink to allow out-of-order delivery and ignore the cache test bits. The MemNet tests reuse the messages of TestMemory.

For complete testing of the multi core system, the same instructions as of the single core system are used, except sw and csr. Some tests of sw instruction might lead to dataraces when multiple cores trying to access the same address, in such cases a core might read data that has already been overwritten. To test csr in multicore, a list of values are passed in the syntax, which acts as source/sink msgs for each core, based on its id.

To compare single and multi threaded benchmarks, a few algorithms such as binary search, complex number multiplication, applying a masked filter to an image, and addition of two vectors were provided. In addition, a quick-sort algorithm to sort a list of values has been implemented. All of these algorithms are first verified for functionality in native x64 systems, compiled for the TinyRV2 processor and then ISA simulated(like Functional Level Verification). The object file is also targeted to the single and the multi core systems that were created using simulators score-sim and mcore-sim that were provided.

The testing strategy used for this lab and for previous labs is to have an incremental approach. We always created basic tests first, then directed tests and finally random tests intended for the test subsystem. The basic tests are straightforward which test the system for functionality, the directed tests are stress tests, corner cases, negative tests, whereas random tests are generally basic and directed tests with random inputs. Different delays are also added to the test source for completeness in verification.

Section 5. Evaluation

We analyzed the performance of our different processors on a variety of benchmark algorithms. These included binary search, complex number multiplication, applying a masked filter to an image, sorting a list, and performing addition between two vectors. We used both single and multithreaded versions of each benchmark. The single threaded sort is a quicksort implementation. The multithreaded sort involves each of the four cores performing quicksort on a quarter of the dataset, and then merging it all back together. The required number of cycles, the CPI, and the number of committed instructions for each benchmark can be seen in Figure 8 and Table 1. It should be noted that for the case where we were running single thread applications on the multi core processor, the simulator would try to run the same program on all cores, affecting the number of instructions committed and the CPI.

While the multicore implementation does provide an improvement in the number of cycles required for these benchmark cases, the improvement does not match the naive expectation of a 4x speed increase over the single core implementation. There may be data dependencies between the different cores, which would stall one or more cores. Additionally there is some overhead involved in distributing the work between each core. This overhead can be seen by the number of cycles for the single threaded benchmarks on the single and multi core processors in Table 1. Depending on the benchmark, this overhead can range from 2.5% additional cycles for binary search to nearly 20% more cycles for complex multiplication. For the multi threaded benchmarks, we see between a 1.8 to 2.75x improvement over the single core implementation. The largest speedup is seen with binary search, as each core can act fully independently, looking over its own portion of the data. Similar speedup is observed in other cases with independent operations, such as complex multiplication and applying a filter. The benefit of multicore is smaller for adding two vectors together, since the number of operations is small enough that the overhead for creating new threads starts to play a large role. The smallest improvement is seen in sorting. This is due to the additional operations that need to be performed to merge the data back together. This adds additional overhead, reducing the benefit of the multicore processor. The multicore design still performs better than the single core implementation at sorting, since the merging step is relatively quick compared to sorting all of the data.

Depending on the complexity of a program and the amount of data dependency, we generally expect the multicore implementation to perform better. For shorter programs, the additional overhead to assign operations to each core will reduce the performance benefit, and potentially negate it for very short programs. The multicore processor, however, is much more complicated than the single core design. It requires many more transistors, as there are three additional cores as well as six additional caches and a complicated network management system. As such, a multicore system will be much more expensive than a single core system. It will require substantially more space on the chip because of these additional transistors. It will also require more energy to power the additional cores and the network. Depending on specifics, the clock frequency of both systems will likely be comparable, as the valid/ready interface between the cores, caches, and network removes these additional structures from the critical path. The clock for the multicore system will be a bit slower, as the increased distance between components will take a bit more time. If the application is set up to be multithreaded, then the multicore processor will perform better than the single core system, since it will be able to execute the program quickly. If the program is not well suited for concurrency, then the multicore processor will be slower, as the additional overhead of assigning a core and the slightly slower clock speed will hinder the performance. Even though the multicore processor requires fewer cycles, this is not a 4x improvement as discussed above, so the reduced number of cycles will not offset the increase in power consumption. We can conclude the multicore processor significantly improves performance of programs with multithreaded nature as it can allocate different threads to different cores to be executed concurrently leading to an increased throughput. The largest benefit of multicore processors is not as substantial when compared to single core processors with the same frequency, but rather in comparison to single core processors with the same performance. To get the same performance, a single core processor would need to be running at 2-3x the frequency of a multicore processor. Power consumption for a processor often goes as the square of the frequency, so for a single core to have the same performance as a multicore processor, it will consume substantially more power.

Section 6. Role and Task Table

Member 1: Peter Wade	**Member 2: Lakshmi Aparna Bolla**	**Member 3: Basant Khalil**
RTL +architect Helped write quicksort algorithm Helped write merge sort algorithm Helped implement the multicore processor design Verifying Connections of Network and Memory	Verification Contributed to sections of the report, such as Testing and The design Helped implement the multicore processor design Contributed to adding more tests and developing a testing strategy	RTL Helped write quicksort algorithm Contributed to Sections of the report, such as the Design Helped write MergeSorting algorithms Helped debug the sorting algorithms

Section 7. Referenced Figures

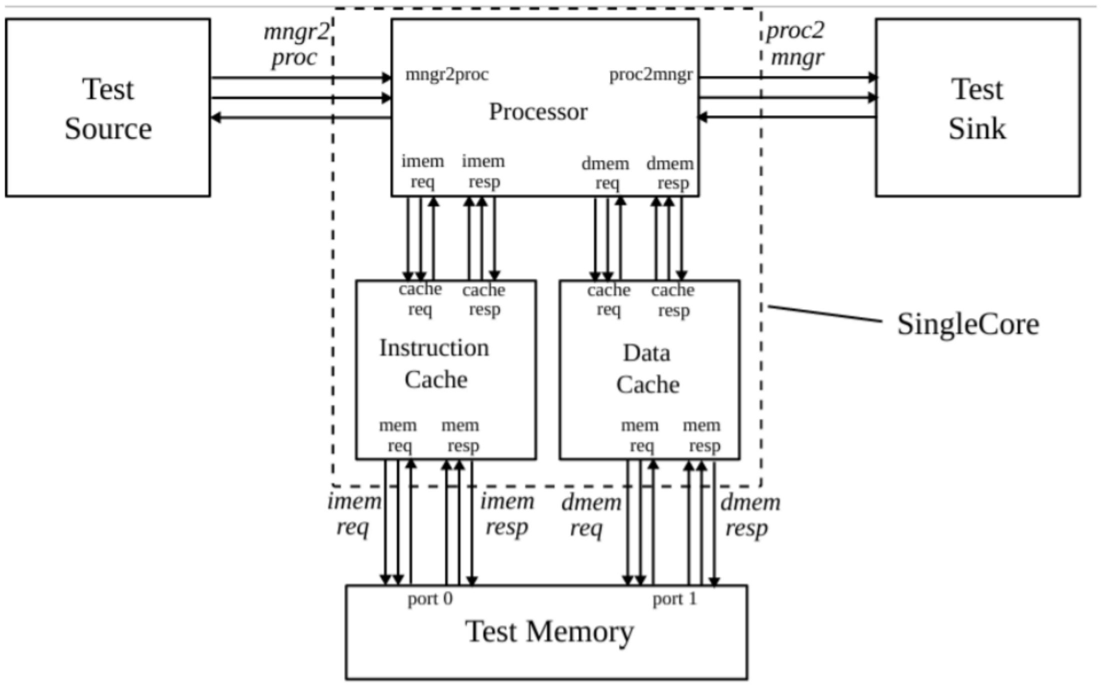

Figure 1: SingleCore – The `SingleCore` module as a whole, is hooked up to the test source, test sink, and test memory for testing and evaluation. Each bundle of the three arrows is a msg/val/rdy port bundle. The ports with inclined names are the top-level ports that `SingleCore` expose.

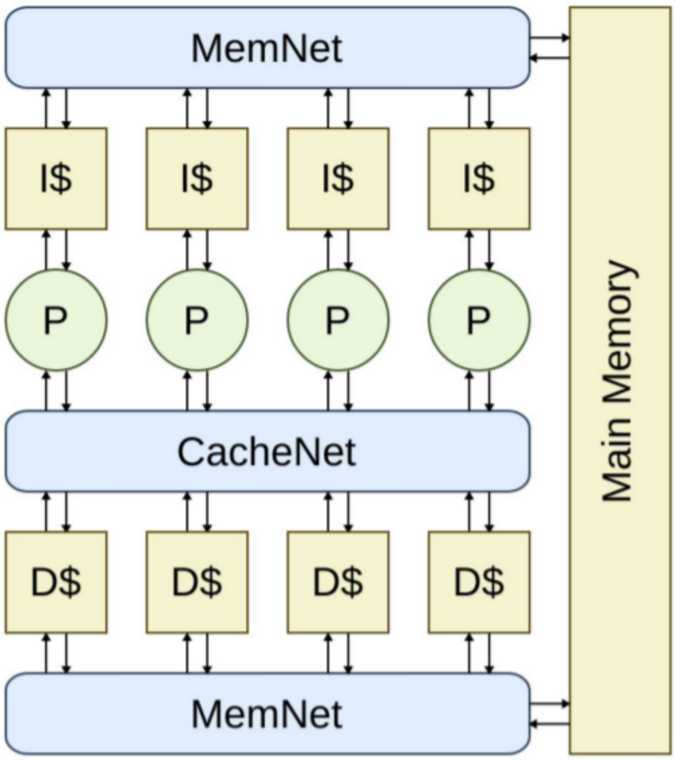

Figure 2: Alternative design block diagram – The alternative design consists of four processors, four private I-caches, a four-banked shared D-cache, and several networks to route dmem request/response from the processors and the D-cache, and refilling both the I-cache and the D-cache. Note that each network shown in the diagram is actually two networks: one for request and the other for response.

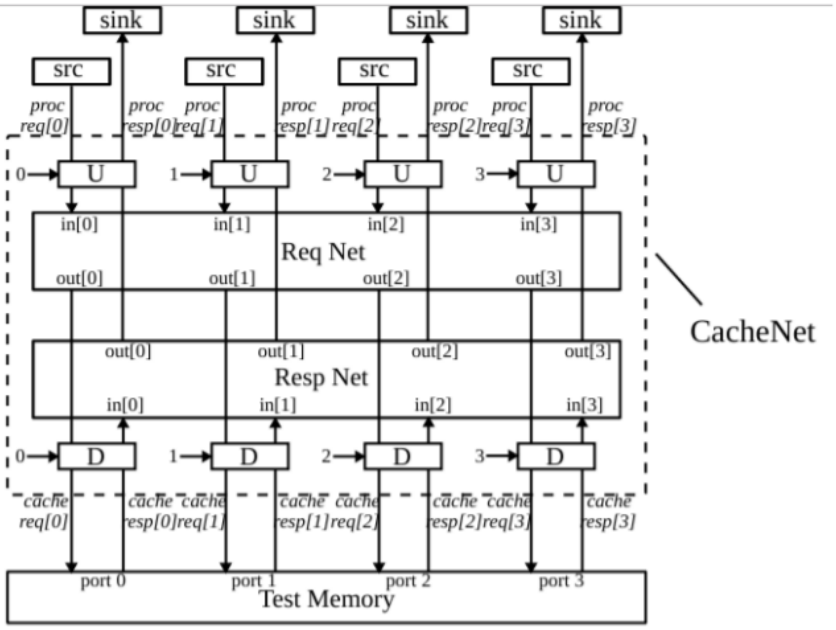

Figure 3: CacheNet – The `CacheNet` module as a whole, is hooked up to the test sources, test sinks, and four-port test memory for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter which takes care of both the request port and the response port, and has its own adapter id. The ports with inclined names are the top-level ports that `CacheNet` exposes.

31 10	9 6	5 4	3 0
tag	index	bank	offset

Figure 4: Memory Address Formats With Banking – Addresses for the data cache include two bank bits which are used to choose which bank to send the memory request; in other words, the bank bits are used as the destination field when converting a memory request message into a network message.

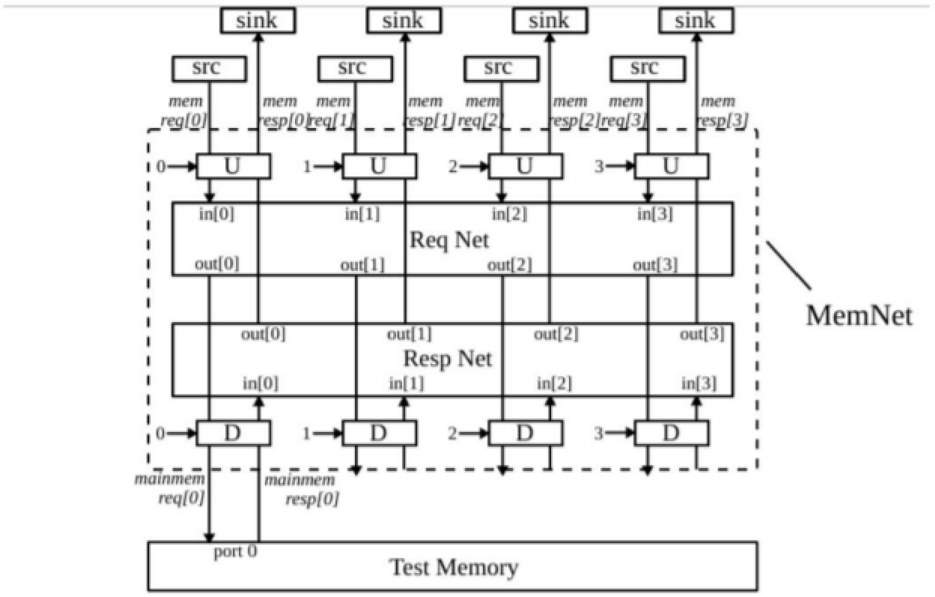

Figure 5: MemNet – The `MemNet` module as a whole, is hooked up to the test sources, test sinks, and single-port test memory for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter which takes care of both the request port and the response port, and has its own adapter id. The ports with inclined names are the top-level ports that `MemNet` exposes.

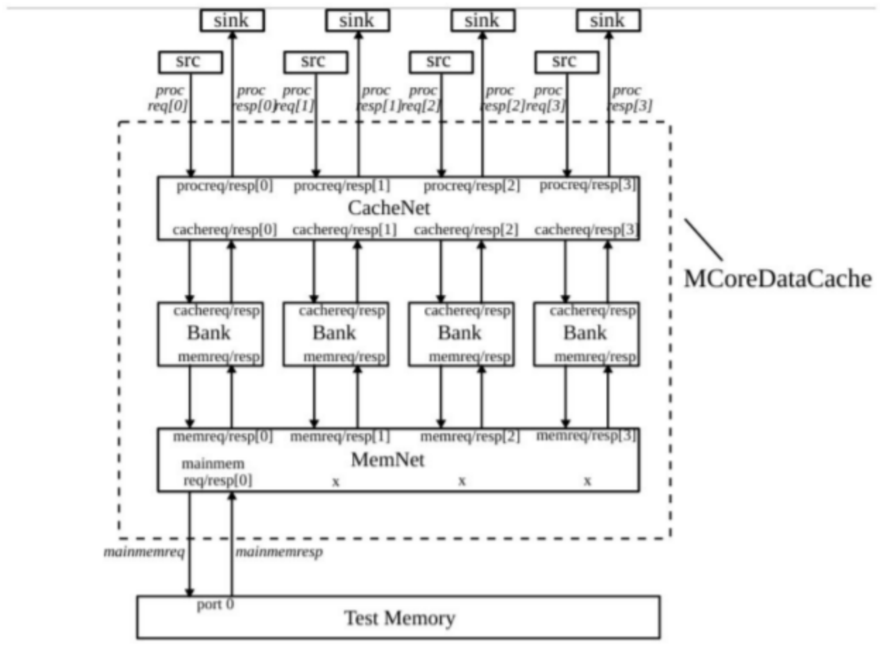

Figure 6: MCoreDataCache – The `McoreDataCache` module as a whole, is hooked up to the test sources, test sinks, and single-port test memory for unit testing. Each arrow is actually a msg/-val/rdy port bundle whose direction is the msg/val direction. We omit the details inside `MemNet` and `CacheNet` but show the port names to match the previous diagrams. The ports with inclined names are the top-level ports that this module exposes.

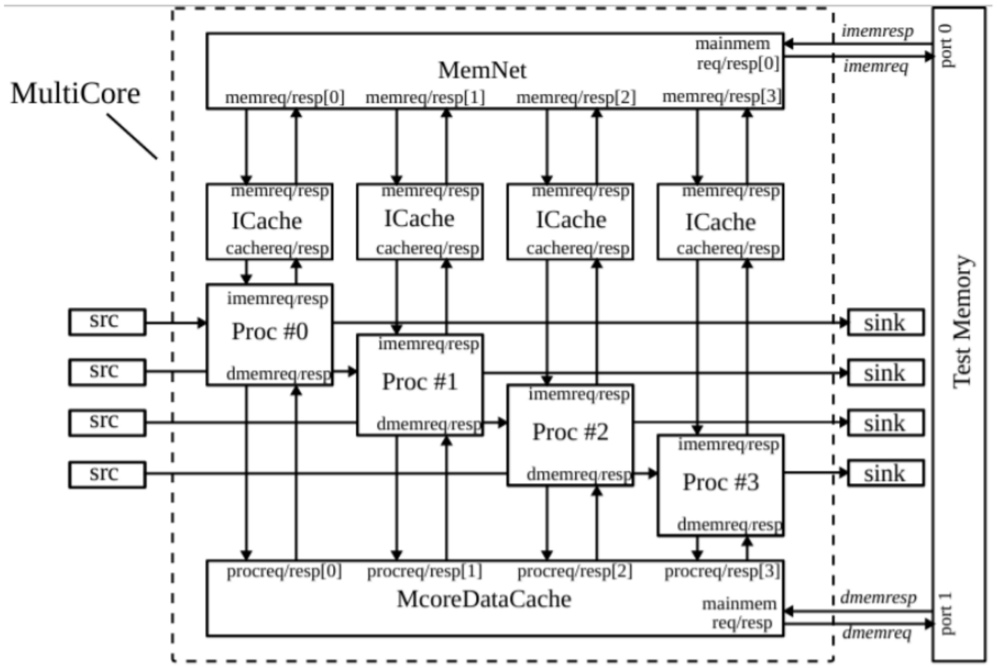

Figure 7: MultiCore – The `MultiCore` module as a whole, is hooked up to a dual-port test memory, four test sources and four test sinks. Note that to not to make the diagram too crowded, we omit the *proc2mngr* and *mngr2proc* port names both at the boundary of `Multicore` as well as in the processor. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. The ports with inclined names are the top-level ports that `MultiCore` exposes.

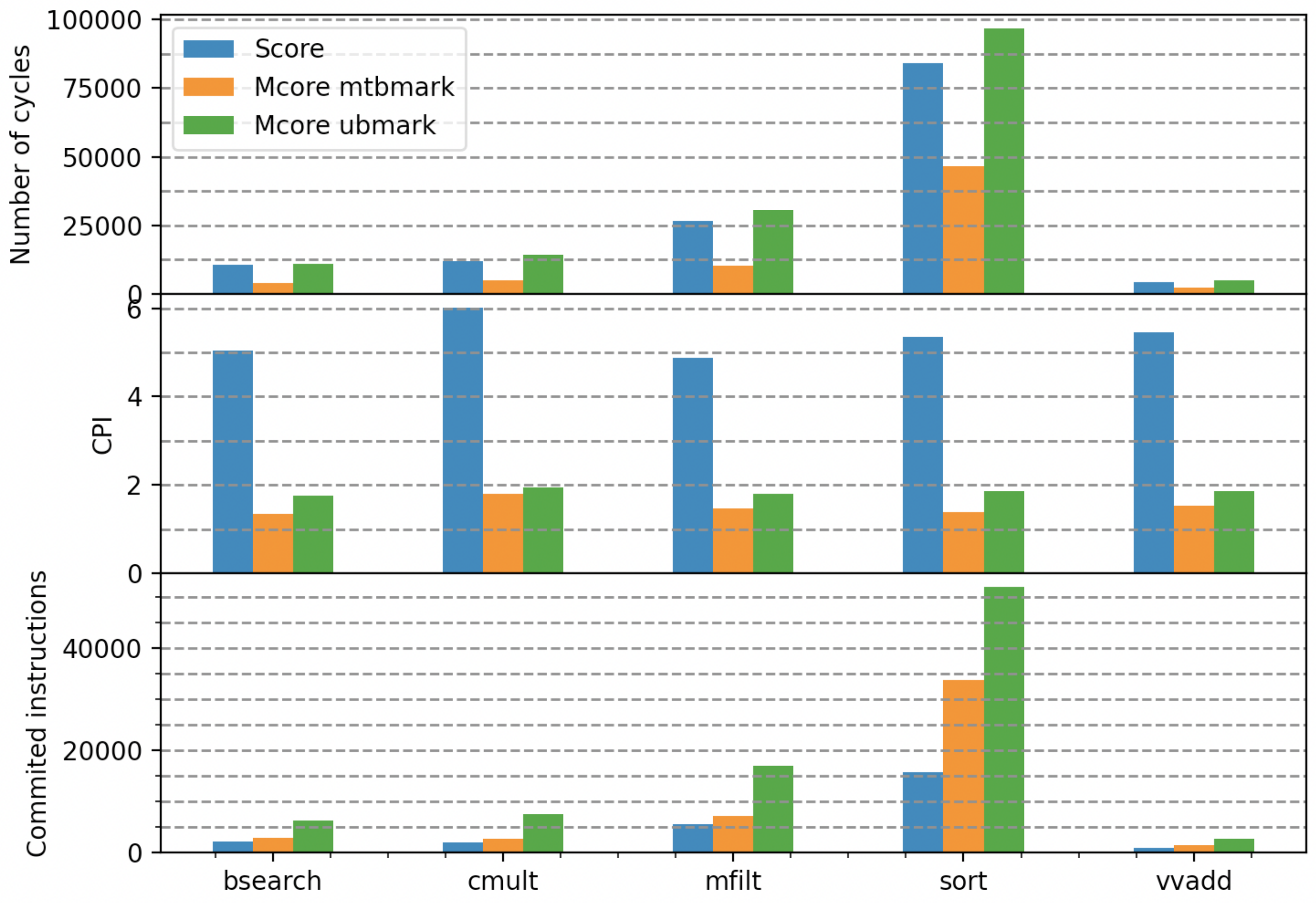

Figure 8: Evaluation data. Score is the data for the single core implementation. Mcore mtbmark is the data for the multi core implementation on the multi threaded benchmarks. Mcore ubmark is the data from the multi core implementation for the single threaded benchmarks.

	Number of Cycles			CPI			Committed instructions			Cycles SCore/ Cycles MCore
	SCore	MCore ubmark	Mcore mtbmark	SCore	MCore ubmark	MCore mtbmark	SCore	MCore ubmark	MCore mtbmark	
bsearch	10628	10898	3849	5.05	1.76	1.35	2106	6194	2848	2.761236
cmult	12097	14411	4881	6.02	1.94	1.80	2011	7415	2715	2.478385
mfilt	26733	30465	10316	4.87	1.80	1.46	5493	16918	7072	2.59141
sort	83922	96816	46484	5.35	1.86	1.38	15692	51998	33688	1.80539
vvadd	4422	4923	2267	5.45	1.85	1.52	811	2656	1495	1.95059

Table 1: The numerical data collected for evaluation.